下载APP

14 | 低耦合：如何快速下线反爬虫系统？

2022-03-02 DS Hunter

《反爬虫兵法演绎20讲》 课程介绍 >

讲述：DS Hunter
时长 17:59 大小 16.48M

你好，我是 DS Hunter，反爬虫专家。又见面了。

前面我们已经详细讲完了反爬虫几乎全部的理论基础，现在我们进入实战看看如何应对真正的爬虫。

由于前面进行了大量的烧脑学习，相信你已经有些疲惫了，我们实战首先找一个轻松的话题：如何快速下线反爬虫系统？

反爬虫需求分析

有同学可能会说：老师你是不是讲错了，我们是来学习如何上线一个反爬虫系统的，为什么这里你说的是下线？

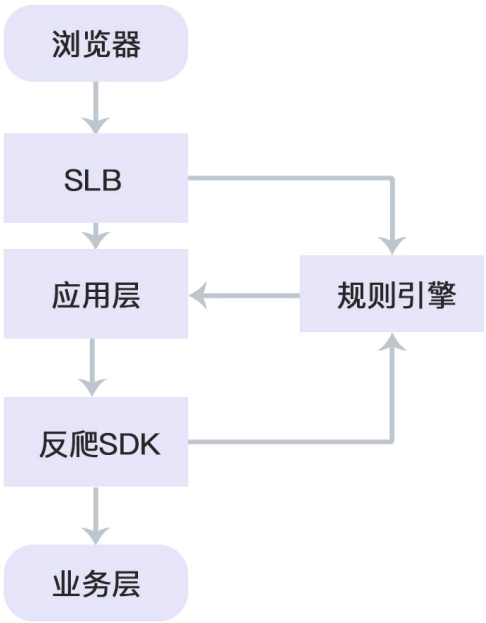
没错，这里我们要讨论的就是如何下线反爬虫系统。原因其实很简单，你可以先想一想，反爬虫需求是永远存在的吗？

很明显，不是的。反爬虫需求来源于爬虫，而爬虫来源于竞对，你有竞对，才有反爬虫需求。

但是，竞对是永远存在的吗？并不是，竞对随时可能消失，甚至被收购。到时候，你的反爬系统将成为系统的累赘，等你离职之后，永远也没有人能把他剥离出去，永远影响着公司的业务。此外，当你变更了规则之后，新的规则如果不适用之前的代码，这些代码还删得掉吗？

因此，**降低反爬虫系统的入侵性，减少与业务代码的耦合，是反爬虫代码的重中之重。**而如果你只知道如何上线，不知道如何下线，那么你的反爬虫项目很可能走不远，后人也无法接手。

既然现在确定了反爬虫系统需要随时被摘除，那么它的设计就可以主要考虑如下两种方式：**依赖注入**和**代理**。我们看一个常规的网络请求：



极客时间

可以看到，我们在右侧直接加了一个规则引擎的模块，用于进行反爬虫的校验。那么，如果这个模块需要随时可以被下掉，他就要做成弱依赖。

例如，和反爬 SDK 耦合，作为一个依赖注入到应用层与业务层之间。通过这样的熔断方式，甚至可以秒级下线反爬虫——你只需要无脑下线规则引擎即可。然后再慢慢摘除后遗症（规则引擎我会在接下来的第 15 课给你详细讲解）。

而如果是和 SLB 耦合，那么就是走代理，让代理托管请求，随时取消代理逻辑即可。

下面我会详细说一下依赖注入系统的设计模式。至于代理模式，就先不展开说了。因为这个就是把所有逻辑做在 SLB，你可以自己试着去实现。这两者没有明确的优劣之分，只是不同人有不同的习惯而已。

实现方式：依赖注入

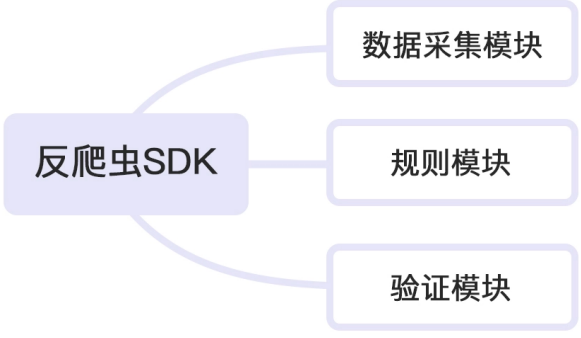
依赖注入设计的大致思路是什么呢？

首先，你要确定你的系统是什么语言的，然后在里面做一个 SDK。接着，再创建一个任意语言的规则引擎，来进行规则检测。最后将两者连接，让 SDK 调用自己的规则引擎。如果规则引擎不返回任何判定结果，那么直接按不是爬虫处理。

确定了设计思路之后，咱们就看一下具体的模块设计吧。

我们假设你的站点是 Node.js 的。（因为前文提到过，一般都是 BFF 扛下了所有，而大部分的 BFF 都是 Node.js。）因此同理，下面整个讲的都是 **Web 反爬虫**，而不是 App。不过，你也可以想一想类似的实现方式。

我们的 SDK 分为以下部分：



极客时间

这三个模块一旦建立，我们依赖注入模式的反爬虫 SDK 系统也就算完成了。下面，我们逐个讨论一下每个模块的建立方式。

数据采集模块

我们先来看数据采集模块。这个数据采集模块就是我们常用的埋点模块，主要用来采集线上用户信息，便于后续做用户特征分析。

考虑到线上主服务系统压力问题，数据采集通常不放在主服务器上。一定要时刻记住这一点： 反爬虫虽然是很重要的一个系统，但是，它不是核心系统，它挂了就挂了，千万不能影响用户正常使用主流程。

因此，一般数据采集会单独建站点来进行采集。这个站点对性能、稳定性各方面都没有什么要求，只要能跑就行。也就是说，在申请资源的时候，不需要申请昂贵的高可用性资源，只需要用最平凡的机器即可，这样可以最大程度地降低成本。成本是反爬虫的一个核心概念，绝对不能忘记，否则很容易失去老板的支持。

好了，既然明确了数据采集模块的核心，接下来我就说说这个核心面临的两大压力吧。采集系统的压力通常来自两块，网络并发和存储压力。我们要在低成本下完成这两样。

为什么这个低优先级的东西会产生高并发？

我们都知道，用户请求页面，请求是有限的。但是如果要进行用户信息统计，这个**需求很容易膨胀，**到最后根本不知道自己要统计多少信息，最终的结果就是并发非常高。所以，即使你的主系统不是高并发的，这个采集系统也可能被迫成为一个高并发的系统。

高并发系统已经成为八股文问题了，甚至连校招生都了如指掌了，这里就不赘述了。但是要注意的是，我们一直在强调成本。高并发有很多为了保证稳定性而付出的高成本，这些都可以舍去。我们要**用稳定性换成本。**

而对于存储压力这个问题，可以求助公司日志系统的人，他们有专业的处理方式，毕竟这个时代，大数据已经不再是一个遥不可及的技术了。还是一样的道理，提需求的时候，首先要强调，我们**对可靠性没有要求**，只要最后能查就行，无需实时，无需可靠，这些都可以舍去，可用就行，否则你的 ROI 很容易算不过来。

所以你会发现，高并发和大数据，两大难题都摆在你面前的时候，千万不要自己去硬着头皮处理，一定要学会使用其余团队的资源，让专业的人替你来处理这些琐事。你的核心竞争力应该是和爬虫对抗。

最终你就得到了一个：**可熔断，非关键，支持高并发，高存储，低成本的集群。**

你可能会说：这不科学啊，我什么都要，结果还什么都得到了，这似乎违背我们做软件的经验。

没错，并不是没有失去什么。仔细回忆下前面的设计，你就会发现，我们失去的是：**稳定性。**也就是说，这个系统在集合了所有的优点之后，还能低成本，主要就是用稳定性换的。不过回头想想，我们刚刚达成了共识——反爬不是一个核心功能，那么就好理解了，这样的代价绝对是超值的。

规则模块

接下来就看规则模块吧。规则模块的逻辑来源于规则引擎。这里我们主要讨论接入问题。至于**低耦合的问题，我们则通过快速下线来实现。**

定位的影响

不过在此之前，你还要额外考虑一个**定位**问题：你在公司的定位，也就是权限，是什么样子的。是只有反爬归你、业务需要和你合作接入反爬，还是业务也归你？这两种情况，接入方向会截然不同。

如果你是反爬方也是业务方，那就是自己开发、自己接入，没有沟通成本，无需跨团队沟通、说服别人，压力会小很多。

而只有反爬虫归你的时候，对于业务方来说，接入反爬，是一个纯成本消耗，没有任何收益。前面我们提到过，反爬不等于安全。接入安全，对于业务方来说，是稳赚的，即使安全最终没有起作用，也有人帮他背锅了，业务会开开心心地接入。 但是反爬不一样，很多公司的反爬组并不承诺拦截所有爬虫，甚至还会勾结业务方放过一些爬虫来达到战略目的（前面提到的随机放过）。

举个简单的例子，你在一个地方部署警察，维持治安，这个地方会很开心的配合。但是如果你部署的是一个特种部队，或者间谍机构，每天神出鬼没，地方也看不出他们在干嘛，甚至可能还要出成本养护，没准出事了，出于保密需求还要提防为这个特种部队背锅，那么其他人一定会怀疑你的。因此，这个行为对于业务方来说就是一个无收益的行为了，他们必然全力阻挠。

所以，如果你本身就是业务方负责人，你可以用行政压力直接命令下去。但是如果你只是反爬组负责人，那么你需要做的就是**降低业务方的接入成本，**不要耗光他们的耐心，否则他们会给出各种不接入反爬的借口。

注入的接入方式

一个最简单的降低成本的方式，就是 SDK 处使用注入的方式来进行接入，对业务方无感。举个例子，业务方引入一个 Maven 包，或者 npm 包，之后就什么也不用做了，这样是理想状况。实际上这可能吗？这并不现实。我们看下前面几讲提过的反爬链路。

第一，用户发起关键服务请求；第二，拦截该请求，转发一个规则引擎的请求，进行规则校验；第三，规则引擎生成 token，带到关键服务请求去，关键服务请求会进行验证，并给出处理。

这个流程走下来你会发现处处有“拦截”，到处都有改代码， 那么如何能无感的进行这个操作呢？

我的建议是，可以考虑如何拦截用户在客户端发起的请求。

我们知道，客户端请求无非就是走 Ajax，自下而上一共有这么几层： XMLHttpRequest，fetch，封装 Ajax 请求的库，比如 jQuery、axios，或者各种框架等等。**拦截越靠上层，越不容易出问题，但是范围越小。**同理，拦截越靠底层，越容易出问题，但是漏网之鱼越少。

我们以难度最高的最底层 XMLHttpRequest 为例。

XMLHttpRequest 最终会走 send 方法，那么只需重写 XMLHttpRequest.prototype.send 即可。

整体的思路是：用户要请求的服务地址，经过检测，发现在关键地址列表里，那么在 prototype 里直接拦截掉，改为调用规则引擎，同时存下原有的 URL。接下来，在规则引擎得到必要的 token 之后，再调用原有的服务，带 token 过去即可。（token 验证在下一个验证模块会讲。）

这样，我们就实现了拦截关键服务，并且可以定制化——毕竟 send 方法归你了，你可以随时添加删除你的 filter。

大概代码逻辑如下：

复制代码

```
const send = XMLHttpRequest.prototype.send;
const open = XMLHttpRequest.prototype.open;
XMLHttpRequest.prototype.send = function(...args){
  try{
    // 对白名单进行处理。
    // 注意递归调用规避
  }
  finally{
    // 这里要避免报错，所以也可以备份原有的args， 避免这里已经被修改。
    send.call(this, ...args);
  }
}
XMLHttpRequest.prototype.open = function(...args){
   try{
    // 对白名单进行处理。
    // 注意递归调用规避
  }
  finally{
    // 这里要避免报错，所以也可以备份原有的args， 避免这里已经被修改。
    oepn.call(this, ...args);
  }
}
```

相信你也看出来了，这样做的优缺点都是非常明显的。缺点显而易见，你动了最底层的 XHR 逻辑， 出问题的概率是最大的。 但是优点也十分明确， 只要是使用 XHR 的库，都被处理了。

而且，回到我开头说的：这个办法是不是下线最快的？当然是。如果你们的竞对被收购了，你只需要在 filter 里面去掉对应的服务地址，就下线了对应的反爬。如果你们收购了所有的竞争对手垄断了市场——当然垄断是不对的，不过这个不在我们的讨论范围内——这

个时候，你只需要移除自己的 JS-SDK，所有的 XHR 没人拦截，自然就回归了正常。因此这个代价是值得的。

但是这里还有一个问题，那就是，反爬是一个非关键功能，如果不小心报错了，你动的又是 XHR 逻辑本身，是不是就意味着整个系统崩溃了？这明显是不能接受的。

因此，你的 js 代码必须非常严格，只要异常一旦出现，就直接切断自己，不要影响主流程。要知道，反爬是随时可以被放弃的，但是主流程不行，主流程是业务核心，是公司的命脉。反爬今天做不了可以明天，明天不行可以后天，总有成功的时候。反爬系统逼急了就自杀，并不是什么丢人的事情。

验证模块

最后这个模块是验证模块。验证模块就是整个反爬的核心了。前面做了这么多的准备工作，就是为了最后的验证。 这里，我们同样采用快速下线的方法来实现低耦合。

这里我还是要提出相同的问题：你是公司的业务方负责人，还是单纯只负责反爬本身？

爬虫验证是误伤的最大来源，如果一个普通用户被误伤了，那么 90% 以上的概率是验证模块出了问题。最终这个责任是定在业务负责人身上的——甚至有的公司反爬组都是保密组，压根在定责系统里查不到。也就是说，业务方有义务，但是没有权利，这个心理落差是任何人都不能承受的。

所以如果你自己就是业务方，自己作死自己承担，没有人会说什么。如果是自己作死、业务承担，那么业务压力会很大，他们迟早会站在你的对立面。那么，为了降低这样的误伤，我们可能在验证时采取一些特殊措施。这些措施在你误伤了用户的情况下，还可以勉强挽回一些损失。

例如，尽可能不拦截用户，而是给予虚假的价格。此外，有意放掉一部分用户，降低误伤概率。你可能会想：放掉用户也会放掉爬虫，这不是为了刷 KPI 而作弊吗？ 非也非也，一定要记住一句话：**反爬从来就不是安全，也不是风控。**

试想一下这样的场景：假设你只拦截 90% 的爬虫，那么爬虫工程师有 9 成的概率刷不到你的反爬请求，他在调试的时候就会自信满满，误以为自己没有命中反爬虫。可是等他的

爬虫真的上了线之后，他的系统一定有 10% 的概率拿到有问题的数据。那么，他的业务方一定不能容忍这种行为。只要有一条数据是错的，被抽到，整批数据业务方都不敢用。

那么他有多大的概率发现自己中反爬了呢？很遗憾，最大值也就只有 10%。这意味着 9 成的情况下，他会被你玩的团团转。 那么，如果你放掉 99% 的爬虫呢？他将只有 1% 的概率发现自己实际上中爬虫了。这个比直接拦截更加恐怖，而且误伤率降低很多。

同时，这种放过爬虫的行为，还可以延伸出多种放过方式，例如根据 IP、区域，以及一些用户维度，设置不同的放过比例，可以玩的花样非常多。

实现方式：代理

这样，三个模块就都讲完了。我们回头看一下，这三个模块构成了你的反爬 SDK， 使用方式其实是一种依赖注入的思想。系统并不包含反爬代码，但是我们的 SDK 通过外部调用的方式把他们都接入了进去。 那么回到开头，除了依赖注入，我们还提了一种方式叫代理对不对？

这里的代理，其实就是你前端的 SLB 直接做代理。与依赖注入的做法一样，你也需要一个系统来做一切。但是在 SLB 一层，会封装价格请求，拦截处理成你的请求。这样做的好处也是一样， 如果有一天发现反爬系统不需要了，那么你只要下线 SLB 上的功能即可。 两种方式都可以，取决于在你的公司内， 哪个更容易部署。 篇幅限制就不展开了。

小结

好了，至今为止，如何低耦合地接入一个反爬系统我们就讲完了。

这一讲我们主要讲了如何使用依赖注入的方式来将反爬系统注入到系统中去，实现快速的下线。依赖注入的系统分为三个模块，数据采集模块、规则模块以及验证模块。当然，数据模块天生就不耦合，而在规则和验证模块，我们选择用快速下线的方式来降低耦合。最后，你也可以自己想想代理这个方式的实现过程。

我想，你应该脑子里就只记住了一个词：下线，或者说低耦合。 因为我们不断地在强调这个事情。如果说记得更多，那应该还记住了一个词：成本。**低耦合和成本，是反爬虫的重中之重。 哪怕是牺牲拦截率，也在所不惜。**这点一定要时刻记在心里。

那么，我们接入的系统，是如何识别爬虫的呢？我们很快进入下一讲：规则引擎。

思考题

最后的最后，又到了愉快的思考题时间。还是老规矩，三选一。

1. 如果在线上，你拦截了你的老板，你的老板大发雷霆，这个时候需要你迅速给出解决方案，根据今天的知识思考一下，你会如何处理呢？
2. 为什么说数据收集是天生无耦合的？他不在系统中吗？
3. 虽然我在文章中说了，依赖注入和代理这两者，是没有明确的优劣之分的，但是，真要比较起来，他们的优缺点分别是什么呢？

可以把想法写在评论区，让我看看你的奇思妙想。 反爬无定势，也许我也可以在评论区学习到更多的思路！

极客时间

叹往昔，举案齐眉何曾齐心协力

恨今日，相濡以沫不如各自纷飞

反爬虫兵法演绎 20 讲
DS Hunter / 反爬虫专家，前某公司研发总监

横批：孩子归我

识别二维码
免费试读

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 13 | 反爬虫基础理论：这是唯一一节严肃的纯理论课

下一篇 15 | 规则引擎：如何快速响应突发的爬虫需求？

精选留言 (1)

写留言

leslie

2022-03-02

代理最大的好处就是下线快和上线方便，注入在"快"和方便这层确实确实有差距；注入的效果可能好些，但是便捷性不如代理，尤其如果Boss是个不懂技术的-代理相对更靠谱；这就回答了第一个问题-万一拦截了Boss的请求怎么办。

作者回复: 代理快是真的快，不过危险性也高一点。